Overview

Last week we covered type definitions, records, datatypes, subtyping.
This week and next week, we will cover additional forms of abstraction:
- polymorphism, type inference
- modules, interfaces
- objects, classes

Today:
- polymorphism and type inference

Consider the humble identity function

A function that returns its input:

def idInt(x: Int) = x
def idString(x: String) = x
def idPair(x: (Int,String)) = x

Does the same thing no matter what the type is.
But we cannot just write this:

def id(x) = x

(In Scala, every variable needs to have a type.)

Another example

Consider a pair “swap” operation:

def swapInt(p: (Int,Int)) = (p._2,p._1)
def swapString(p: (String,String)) = (p._2,p._1)
def swapIntString(p: (Int,String)) = (p._2,p._1)

Again, the code is the same in both cases; only the types differ.
But we can’t write

def swap(p) = (p._2,p._1)

What type should \( p \) have?
Another example

Consider a higher-order function that calls its argument twice:

```scala
def twiceInt(f: Int => Int) = {x: Int => f(f(x))}
def twiceStr(f: String => String) = {x: String => f(f(x))}
```

Again, the code is the same in both cases; only the types differ.

But we can’t write

```scala
def twice(f) = {x => f(f(x))}
```

What types should \( f \) and \( x \) have?

Type parameters

In Scala, function definitions can have type parameters

```scala
def id[A](x: A): A = x
```

This says: given a type \( A \), the function \( \text{id}[A] \) takes an \( A \) and returns an \( A \).

```scala
def swap[A,B](p: (A,B)): (B,A) = (p._2,p._1)
```

This says: given types \( A, B \), the function \( \text{swap}[A,B] \) takes a pair \( (A,B) \) and returns a pair \( (B,A) \).

```scala
def twice[A](f: A => A): A => A = {x:A => f(f(x))}
```

This says: given a type \( A \), the function \( \text{twice}[A] \) takes a function \( f: A => A \) and returns a function of type \( A => A \)

Polymorphism: More examples

Polymorphism is even more useful in combination with higher-order functions.

Recall \( \text{compose} \) from the lab:

```scala
def compose[A,B,C](f: A => B, g: B => C) = {x:A => g(f(x))}
```

Likewise, the \( \text{map} \) and \( \text{filter} \) functions:

```scala
def map[A,B](f: A => B, x: List[A]): List[B] = ...
def filter[A](f: A => Bool, x: List[A]): List[A] = ...
```

(though in Scala these are usually defined as methods of \( \text{List}[A] \) so the \( A \) type parameter and \( x \) variable are implicit)

Scala’s type parameters are an example of a phenomenon called polymorphism (= “many shapes”)

More specifically, parametric polymorphism because the function is parameterized by the type.

- Its behavior cannot “depend on” what type replaces parameter \( A \).
- The type parameter \( A \) is abstract

We also sometimes refer to \( A, B, C \) etc. as type variables
Formalization

- We add type variables $A, B, C, \ldots$, type abstractions, type applications, and polymorphic types:
  \[
  e ::= \cdots | \Lambda A. e | e[\tau]
  \]
  \[
  \tau ::= \cdots | A | \forall A. \tau
  \]

- We also use (capture-avoiding) substitution of types for type variables in expressions and types.

- The type $\forall A. \tau$ is the type of expressions that can have type $\tau[\tau'/A]$ for any choice of $A$. ($A$ is bound in $\tau$.)

- The expression $\Lambda A. e$ introduces a type variable for use in $e$. (Thus, $A$ is bound in any type annotations in $e$.)

- The expression $e[\tau]$ instantiates a type abstraction.

- Define $L_{\text{Poly}}$ to be the extension of $L_{\text{Data}}$ with these features.

Formalization: Types and type variables

- Complication: Types now have variables. What is their scope? When is a type variable in scope in a type?

- The polymorphic type $\forall A. \tau$ binds $A$ in $\tau$.

- We write $FTV(\tau)$ for the free type variables of a type:
  \[
  FTV(\tau) = \{ A \} \quad FTV(\tau_1 \times \tau_2) = FTV(\tau_1) \cup FTV(\tau_2) \\
  FTV(\tau_1 + \tau_2) = FTV(\tau_1) \cup FTV(\tau_2) \\
  FTV(\forall A. \tau) = FTV(\tau) - \{ A \} \\
  FTV(\tau) = \emptyset \quad \text{otherwise}
  \]

- $FTV(x_1:\tau_1, \ldots, x_n:\tau_n) = FTV(\tau_1) \cup \cdots \cup FTV(\tau_n)$

- Alpha-equivalence and type substitution are defined similarly to expressions.

Formalization: Typechecking polymorphic expressions

- Idea: $\Lambda A. e$ must typecheck with parameter $A$ not already used elsewhere in type context

- $e[\tau_0]$ applies a polymorphic expression to a type. Result type obtained by substituting for $A$.

- The other rules are unchanged

Formalization: Semantics of polymorphic expressions

- To model evaluation, we add type abstraction as a possible value form:
  \[
  v ::= \cdots | \Lambda A. e
  \]

- with rules similar to those for $\lambda$ and application:

- In $L_{\text{Poly}}$, type information is irrelevant at run time.

(Other languages, including Scala, do retain some run time type information.)
Convenient notation

- We can augment the syntactic sugar for function definitions to allow type parameters:

  let fun f[A]:τ(x) = e in ...

- This is equivalent to:

  let f = ΛA. λx : τ. e in ...

- In either case, a function call can be written as

  \( f[\tau](x) \)

Examples in L_Poly

- Identity function

  \( id = ΛA.λx:A. x \)

- Swap

  \( swap = ΛA.ΛB.λx:A × B. (snd x, fst x) \)

- Twice

  \( twice = ΛA. λf : A → A. λx:A. f(f(x)) \)

- For example:

  \( swap[int][str](1,"a") ⇓ ("a",1) \)

  \( twice[int](λx: 2 × x)(2) ⇓ 8 \)

Examples, typechecked


Lists and parameterized types

- In Scala (and other languages such as Haskell and ML), type abbreviations and definitions can be parameterized.

- List[_] is an example: given a type T, it constructs another type List[T]

  \( \text{deftype List[A] = [Nil : unit; Cons : A × List[A]]} \)

- Such types are sometimes called type constructors

  (See tutorial questions on lists)

- We will revisit parameterized types when we cover modules
Other forms of polymorphism

- Polymorphism refers to several related techniques for “code reuse” or “overloading”
  - Subtype polymorphism: reuse based on inclusion relations between types.
  - Parametric polymorphism: abstraction over type parameters
  - Ad hoc polymorphism: Reuse of same name for multiple (potentially type-dependent) implementations (e.g. overloading + for addition on different numeric types, string concatenation etc.)

- These have some overlap
- We will discuss overloading, subtyping and polymorphism (and their interaction) in future lectures.

Hindley-Milner type inference

- As seen in even small examples, specifying the type parameters of polymorphic functions quickly becomes tiresome
  - \( \text{swap}[^{\text{int}}][^{\text{str}}] \quad \text{map}[^{\text{int}}][^{\text{str}}] \quad \ldots \)

- Idea: Can we have the benefits of (polymorphic) typing, without the costs? (or at least: with fewer annotations)
- Type inference: Given a program without full type information (or with some missing), infer type annotations so that the program can be typechecked.

Hindley-Milner example [Non-examinable]

- A very influential approach was developed independently by J. Roger Hindley (in logic) and Robin Milner (in CS).

- Idea: Typecheck an expression symbolically, collecting “constraints” on the unknown type variables
- If the constraints have a common solution then this solution is a most general way to type the expression
  - Constraints can be solved using unification, an equation solving technique from automated reasoning/logic programming
- If not, then the expression has a type error

\[
\vdash \lambda x : A. (\text{snd}, \text{fst} \ x) : B
\]

\( A, B \) are the as yet unknown types of \( x \) and \( \text{swap} \).
As an example, consider `swap` defined as follows:

\[ \vdash \lambda x : A. (\text{snd } x, \text{fst } x) : B \]

\( A, B \) are the as yet unknown types of \( x \) and \( \text{swap} \).

- A lambda abstraction creates a function: hence \( B = A \rightarrow A_1 \) for some \( A_1 \) such that \( x : A \vdash (\text{snd } x, \text{fst } x) : A_1 \)
- A pair constructs a pair type: hence \( A_1 = A_2 \times A_3 \) where \( x : A \vdash \text{snd } x : A_2 \) and \( x : A \vdash \text{fst } x : A_3 \)
- This can only be the case if \( x : A_3 \times A_2 \), i.e. \( A = A_3 \times A_2 \).

As an example, consider `swap` defined as follows:

\[ \vdash \lambda x : A. (\text{snd } x, \text{fst } x) : B \]

\( A, B \) are the as yet unknown types of \( x \) and \( \text{swap} \).

- A lambda abstraction creates a function: hence \( B = A \rightarrow A_1 \) for some \( A_1 \) such that \( x : A \vdash (\text{snd } x, \text{fst } x) : A_1 \)
- A pair constructs a pair type: hence \( A_1 = A_2 \times A_3 \) where \( x : A \vdash \text{snd } x : A_2 \) and \( x : A \vdash \text{fst } x : A_3 \)
- This can only be the case if \( x : A_3 \times A_2 \), i.e. \( A = A_3 \times A_2 \).
- Solving the constraints: \( A = A_3 \times A_2, A_1 = A_2 \times A_3 \) and so \( B = A_3 \times A_2 \rightarrow A_2 \times A_3 \).
Let-bound polymorphism [Non-examinable]

- An important additional idea was introduced in the ML programming language, to avoid the need to explicitly introduce type variables and apply polymorphic functions to type arguments.
- When a function is defined using `let fun` (or `let rec`), first infer a type:

\[
\text{swap} : A_3 \times A_2 \rightarrow A_2 \times A_3
\]

- Then abstract over all of its free type parameters.

\[
\text{swap} : \forall A, \forall B. A \times B \rightarrow B \times A
\]

- Finally, when a polymorphic function is applied, infer the missing types.

\[
\text{swap}(1, "a") \mapsto \text{swap}[\text{int}][\text{str}](1, "a")
\]

ML-style inference: strengths and weaknesses

- **Strengths**
  - Elegant and effective
  - Requires no type annotations at all
- **Weaknesses**
  - Can be difficult to explain errors
  - In theory, can have exponential time complexity (in practice, it runs efficiently on real programs)
  - Very sensitive to extension: subtyping and other extensions to the type system tend to require giving up some nice properties
  - (We are intentionally leaving out a lot of technical detail — HM type inference is covered in more detail in ITCS.)

Type inference in Scala

- Scala does not employ full HM type inference, but uses many of the same ideas.
- Type information in Scala flows from function arguments to their results.

```scala
def f[A](x: List[A]): List[(A,A)] = ...
```

```scala
f(List(1,2,3)) // A must be Int, don't need f[Int]
```

- and sequentially through statement blocks.

```scala
var l = List(1,2,3); // l: List[Int] inferred
def map[A](f: A => B, l: List[A]): List[B] = ...
```

```scala
scala> map({x: Int => x + 1}, List(1,2,3))
res0: List[Int] = List(2, 3, 4)
```

- Type information does **not** flow across arguments in the same argument list.

```scala
scala> map({x => x + 1}, List(1,2,3))
<console>:25: error: missing parameter type
```

- But it **can** flow from earlier argument lists to later ones:

```scala
def map2[A](l: List[A])(f: A => B): List[B] = ...
```

```scala
scala> map2(List(1,2,3)) {x => x + 1}
res1: List[Int] = List(2, 3, 4)
```
Compared to Java, many **fewer** annotations needed
Compared to ML, Haskell, etc. many **more** annotations needed
The reason has to do with Scala’s integration of **polymorphism** and **subtyping**
- needed for integration with Java-style object/class system
- Combining subtyping and polymorphism is tricky (type inference can easily become undecidable)
Scala chooses to avoid global constraint-solving and instead propagate type information **locally**

**Today we covered:**
- The idea of thinking of the same code as having many different types
- Parametric polymorphism: makes the type parameter explicit and abstract
- Brief coverage of **type inference**.

**Next time:**
- Programs, modules, and interfaces